Object Relater *Plus*

Object Database Manipulation Language (ODML) User's Guide

Release 3.1

Copyright © 1995 1996 1999 2008 Bryon K. Ehlmann

1 Introduction

Object Relater *Plus* (OR+) is a tool for developing object databases (ODBs) [5]. It allows relationships between objects to be more easily implemented and maintained, plus it provides other features, like persistence and ODMG-93 compatibility. OR+ is being developed at Florida State and Florida A&M Universities and Southern Illinois University Edwardsville to facilitate the development of scientific databases and tools and to research and experiment with ODB enhancements [2, 3]. Object persistence is provided in OR+ using Object Store [6].

The Object Database Manipulation Language (ODML) of OR+ is used to retrieve and modify objects in an ODB, which must be defined by the Object Database Definition Language (ODDL), as described in the Object Relater *Plus* ODDL User's Guide. ODML provides generic operations on ODDL-defined objects that are compatible with the ODMG-93 C++ OML (Future Binding) [1]. In addition, support is provided for the Object Relationship Notation (ORN) [4].

Using ODML, a database can be created, opened, and closed. Transactions on a database can be started, committed, and aborted. Within a transaction, database objects can be created, modified, deleted, and relationships between the objects can be created, changed, and destroyed. Object deletion and relationship change and destruction are not merely primitive object operations, rather they are higher-level operations that treat objects as complex and composite objects.

This document illustrates ODML via sample programs that create, access, and update a company database. The ORD for this database is shown in Fig. 1, and a partial ODDL specification is shown in Fig. 2. The Appendix provides precise definitions and documentation for the enums, macros, object classes, and member functions that constitute ODML.

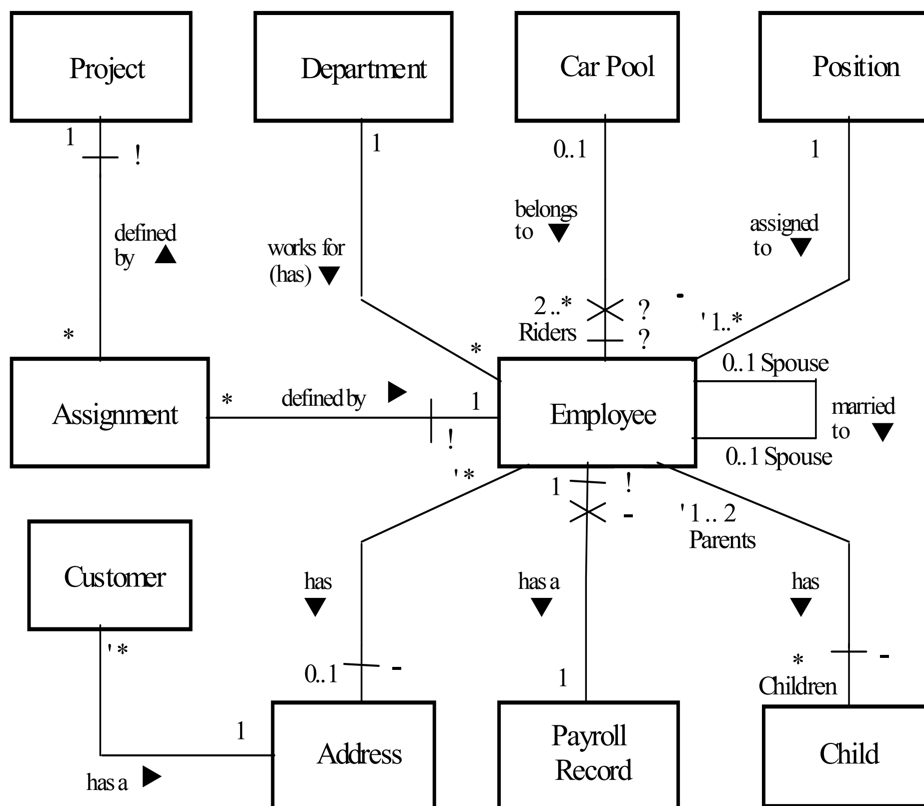

Fig. 1. Example ORD for the ODDL in Fig. 4.

```
Database CompanyDb  // Company DataBase

$sidebar 1
derived  // attribute is derived
$.
{
   ...
   class employee {
      d_String         SSN;       // Social Security Number
      d_String         Name;      // Last name, First name
      d_Date           BirthDate;
      address          Address inverse Employees '<*-to-0..1>|-;
      payroll_record   PayrollRecord inverse Employee |!X-<1-to-1>;
      department       Dept inverse Employees <*-to-1>;
      position         Position inverse Employee '<1..*-to-1>;
      Set<assignment>  Assignments inverse Employee |!<1-to-*>;
      employee         Spouse inverse Spouse <0..1-to-0..1>;
      List<child>      Children inverse Parents '<1..2-to-*>|-;
      car_pool         CarPool inverse Riders |?X?<2..-to-0..1>;
$C++
      void RaiseSalary(int percentage);
      void Print();
      ...
$.
   };
   extent Set<employee>  Employees  key SSN; [1]

   class department {
      d_String         Name;
      d_String         Location;    // Building No.
      Set<employee>    Employees inverse Dept; [2]
                          // <1-to-*> based on inverse
$C++
      void Print();
      ...
$.
   };
   extent Set<department>  Departments  key Name;

   class child X- {
      d_String         Name;
      Set<employee>    Parents inverse Children |-<*-to-1..2>';
      ...
   };

   class assignment {
      project          Project  inverse Assignments;
      employee         Employee inverse Assignments;
      Date             StartingDate;
      Date             TerminationDate;
      int              Duration; |1  // in days
      ...
   };
   extent Set<assignment>  Assignments  key (Project, Employee);
...
};
...
$footnotes
[1] index on key, ordered index on (Dept, Name)
[2] ordered index on Name
...
$.
```

Fig. 2. Partial ODDL for a Company Database.

```
// employee.C
#include <iostream.h>
#include "employeeOR+.C"
   // Above file is generated by ODDL.
employee::employee(char* name) : Name(name)
{ Employees_Ptr->insert_element(this); }

employee::~employee() { ::Employees_Ptr->remove_element(this); }

void employee::Print()
{
    cout << "   " << Name;
    if (Dept)
       cout << "    -> " << Dept->Name;
    else
       cout << "    -> NULL";
    cout << endl;
}
```

```
1.  // applic_prog1.C
2.  #include <iostream.h>
3.  #include "employeeOR+.h"
4.  #include "departmentOR+.h"
5.      // Above <class-name>OR+.h files are generated by ODDL and must be
6.      //    included in application programs that references these classes.
7.  void main()
8.  {
9.     Database d;
10.    Transaction t1, t2;
11.    d.create("co1", 0644);
12.    t1.begin();
13.    Departments_Ptr = NEW(d, d_Set<department*>);
14.    Employees_Ptr = NEW(d, d_Set<employee*>);
15.    d.set_object_name(Employees_Ptr, "Employees");
16.    d.set_object_name(Departments_Ptr, "Departments");
17.    employee* e1 = NEW(d, employee)("Doe, John");
18.    employee* e2 = NEW(d, employee)("Howard, Bobby");
19.    employee* e3 = NEW(d, employee)("Greene, Fred");
20.    employee* e4 = NEW(d, employee)("Williams, Yvette");
21.    employee* e5 = NEW(d, employee)("Kelly, Viola");
22.    employee* e6 = NEW(d, employee)("Johnson, Kate");
23.    department* d1 = NEW(d, department)("Accounting");
24.    department* d2 = NEW(d, department)("Marketing");
25.    department* d3 = NEW(d, department)("Personnel");
26.    // Create additional objects and set attributes.
27.    ...
28.    t1.commit();
29.    d.close();
30.    d.open("co1");
31.    t2.begin();
32.    Departments_Ptr = (d_Set<department*>*)d.lookup_object("Departments");
33.    Employees_Ptr = (d_Set<employee*>*)d.lookup_object("Employees");
34.    ...  // Access and manipulate database.
35.    t2.commit();
36.    d.close();
37.    d.open("co1");
38.    t2.begin();
39.    d.destroy();
40. }
```

Fig. 3. Sample Company Program that Creates and Accesses Company Database including Source File for Employee Class Function Member Bodies

2 Database

This section documents the operations that can be performed on an ODDL defined database. A brief description is given for each operation, which is defined as a member function, with a more detailed description given in the Appendix. Most of the operations are demonstrated in **`applic_prog1.C`** in Fig. 3. This program creates (and opens) a company database, creates some named extent objects providing roots into the database, creates some employee objects, and then

closes the database. It then opens the database to access and manipulate its contents, closes it again, and finally opens it and destroys it.

Instantiation and System Initialization

A database is declared and the database system is initialized, if not already initialized, by the declaration

`Database` *database-name*`;`

Line 9 in Fig. 3 declares a Database object identified internally within the program as **`d`**. A database declaration initializes the database with access status **`not_open`**.

The access status of a database is **`not_open`**, **`read_write`**, **`read_only`**, or **`exclusive`**. **`not_open`** means that objects cannot be created or referenced in the database. **`read_write`** means the user can create, reference, and manipulation objects in the database. **`read_only`** means the user can only reference or access objects and attributes in the database. **`Exclusive`** means that the user has exclusive use of the database.

`create()`

The create operation on a Database associates an externally named database with the object, creates an empty version of this database, assigns access permissions to the database, and opens it with status **`read_write`**. The permissions are identical to those that can be assigned to Unix files via the **`chmod`** command. Line 11 in Fig. 3 associates the database **`co1`** with **`d`**, creates an empty version of **`co1`**, and opens it with status **`read_write`**.

The create operation can be made to implicitly destroy a database if it already exists. See Appendix.

`open()`

The open operation on a Database object associates an externally named database with the object and opens this database with the given status or with the default status of **`read_write`**. Lines 30 and 37 in Fig. 3 associates the database **`co1`** with **`d`** and opens it with status **`read_write`**. If the database **`co1`** were not found, an exception would occur.

An open operation can be made to optionally create a database if the database does not already exists. See Appendix.

`close()`

Closing a database, sets its status to **`not_open`**. Database objects can no longer be referenced. Lines 29 and 36 in Fig. 3 close the database "co1".

`destroy()`

The destroy operation on a Database object deletes the external database associated with the object. The database must be opened and the operation must be done inside of a transaction. The effects of the destroy operation cannot be undone by aborting the transaction. Line 39 in Fig. 3 deletes the database **`co1`**.

`set_object_name()`

This operation provides a name for an object within the database. A named object provides an entry point into the database. Objects may be named twice; however, two different objects may not be named the same. Lines 15 and 16 in Fig. 3 demonstrate the use of this operation. In line 15, the object referenced by **`Employee_Ptr`**, which happens to be an extent, is named **`Employees`**.

`lookup_object()`

This operation searches for objects by name. If the object is found, a pointer to the object is returned, and an entry point into the database is established. On the other hand, if the object is not found, the returned pointer value is 0. Pointers to named objects (or to database objects in general) are not preserved across transactions, i.e., objects must be relooked up in a subsequent transaction. Lines 32 and 33 in Fig. 3 demonstrate this operation. Line 32 searches for the object named **`Departments`**, which is the extent for class **`department`**, and casts the returned pointer value appropriately for assignment to **`Department_Ptr`**.

3 Transactions

A transaction defines an atomic sequence of updates to a database that either all take place or are all undone. Transaction objects must be explicitly created and begun to access, create, modify, and delete database objects. Transaction objects are not persistent. A number of operations can be performed on a Transaction object.

The program in Fig. 3 illustrates some of these operations. The declaration in line 10 declares two transactions, **t1** and **t2**. These transactions are created as non-nested. The declaration below, on the other hand, would create a nested transaction **tn** with parent transaction **t1**.

```
Transaction tn(&t1);
```

With a nested transaction, the outer transaction directly affects the inner transaction. If there are changes made in the inner transaction and the outer transaction does not commit the inner changes will not be committed. In addition, if the inner transaction does commit, but the outer transaction is aborted, then all inner transaction modifications are aborted as well. If the inner transaction aborts, previous changes made in the outer transaction are not affected.

The **CurrTransPtr()** operation simply returns a pointer to the current transaction.

The **begin()** operation begins a transaction. Line 12 in Fig. 3 begins transaction **t1**.

The **commit()** operation commits all object creations, modifications, and deletions to the database which were made since the beginning of the transaction. This operation releases any locks held by the transaction, along with setting embedded pointers in non-persistent objects to 0. A number of relationship constraints are checked during a commit operation, and exceptions result if these constraints are not satisfied. **commit()** does not delete the transaction object. Line 28 in Fig. 3 commits transaction **t1**.

The **abort()** operation releases all locks and aborts all changes made within the transaction. The statement **t1.abort();** would abort transaction **t1** if given prior to line 28 and would result in an empty database.

SetRXCmode() sets the *Relationship eXChange mode*, and **ResetRXCmode()** resets the mode. **RXCmode()** returns true if the transaction is in RXC mode.. The default is that a transaction is not in RXC mode.

Sometimes it is necessary to suspend lower bound multiplicity checks and resultant actions, i.e., implicit deletes, that would normally result from explicit relationship destructions to allow relationships among existing objects to be reshuffled. For example, assume the relationship between employees and payroll records was |**!<1-to-1>**, instead of |**!X-<1-to-1>**. this permits relationships between employees and payroll records, once established, to be changed. Suppose that the payroll records assigned to three current employees must be reshuffled—an very unlikely application with respect to this relationship but perhaps not so for others. Employee e1 currently has payroll record r1, e2 has r2, and e3 has r3. r1 must be reassigned to e2, r2 to e3, and r3 to e1. This can be accomplished by the following code within a Transaction t:

```
t.SetRXCmode();
e1.PayrollRecord = e2.PayrollRecord = e3.PayrollRecord = 0;
e1.PayrollRecord = &r3;
e2.PayrollRecord = &r1;
e3.PayrollRecord = &r2;
t.ResetRXCmode();
```

The function **SetRXCmode()** above sets the RXC mode for the transaction. The function **ResetRXCmode()** resets this mode. When RXCmode is set, implicit deletes that would normally result from explicit relationship destructions are not done; however, relevant attributes, e.g., **PayrollRecord** above, are still checked at transaction commit to ensure that lower bound multiplicities are satisfied. Exceptions result if they are not. Without the RXC mode, certain manipulations on some relationship types, such as those above, could not be done.

4 Objects

This section discusses the generic operations on objects that are provided by ODML, namely creation and deletion.

Creation

Objects are created, using the **new** operator in ODML. This overloaded operator not only allocates free store for an object but also accept arguments that determine its lifetime. The new operator creates a new object in the given database of a given type. The macro **NEW** simplifies the invocation of the **new** operator. In Fig. 3, line 13, **Departments_Ptr** is set to point to allocated space for a **d_Set** of **department*** pointers. This space is not only accessible in the program but is made persistent in the database associated with **d**, with here is **co1**. The **employee*** pointers on lines 17 - 22 are set to point to individual employee instances that are created in database **co1** via the macro **NEW**.

Delete()

The **Delete()** operation deletes an object from the database. The object may be a complex object. That is, its deletion may result in the implicit deletion of related objects due to its relationships to these objects as discussed in Section 6. In Fig. 4, line 28 illustrates the deletion of an employee object referenced by **pe**.

The sample program in Fig. 4 again illustrates some of the classes, macros, and functions explained thus far and will be used to illustrate additional ODML constructs. The program adds a public relations department to the company database and an employee, who is assigned to this new department. In addition, it deletes all temporary employees and reassigns all employees in customer support, except for the department's manager, to the public relations department. It then prints all departments, including their employees. The following paragraphs will discuss this program in more detail.

```
1.  // applic_prog2.C
2.  #include <iostream.h>
3.  #include "CompanyDbOR+.h"
4.  void PrintDepts()   // Print all departments.
5.  {
6.    cout << "Company Employees by Department:" << endl;
7.    d_Iterator<department*> d(*Departments_Ptr);
8.    department* dp;
9.    while (d.next(dp))
10.       dp->Print();    // Print department info, including employees.
11.  }
12.  void main()
13.  {
14.      Database db;
15.      Transaction t;
16.      db.open("co1");
17.      t.begin();
18.      Departments_Ptr = (d_Set<department*>*)db.lookup_object("Departments");
19.      Employees_Ptr = (d_Set<employee*>*)db.lookup_object("Employees");
20.      department* pd = NEW(db, department)("Public Relations");
21.      pd->Location = "205 Building B";
22.      employee* pe = NEW(db, employee)("Doe, John Q.");
23.      pe->Dept = pd;     //or pd->Employees.insert_element(pe);
24.      ...  // Create payroll_record object and set additional attributes.
25.      d_Iterator<emp*> e(*Employees_Ptr);
26.      while (e.next(pe))
27.         if (pe->Position->Type == Temporary)
28.            pe->Delete()
29.         else
30.            if (pe->Dept->Name == "Customer Support" &&
31.                pe->Position->Name != "Department Manager")
32.                pe->Dept = pd;
33.      PrintDepts();
34.      t.commit();
35.      db.close();
36.  }
```

Fig. 4. Sample Company Database Application Program

The two named objects, **Departments** and **Employees**, provide entry points into the declared database (lines 18 and 19). The pointer variables that reference the named objects, **Department_Ptr** and **Employees_Ptr**, are declared in **CompanyDbOR+.h**, which is generated by ODDL. Invocations of the **NEW** macro in lines 20 and 22 create two new persistent objects, department **Public Relations** and employee **John Q. Doe**, respectively. Line 21 sets the attribute **Location** for department **Public Relations**.

5 Collections and Iterators

A Collection is an abstract class whose objects are collections of elements. A Collection, itself, cannot have instances, but instances can occur in concrete classes derived from a Collection, e.g. Set and List. A Set is an unordered collection of unique elements. All operations on Collections are inherited by Set. A List, on the other hand, is an ordered collection that may or may not allow duplicate elements. In ODML, a Collection, Set, and List map to a d_Collection, d_Set, and d_List, respectively; however, for object-valued attributes with inverse clauses, these later classes are wrapped in generated C++ classes that function the same but provide the additional relationship semantics.

A d_Iterator is used to access a Collection. Iterators use a consistent protocol, to sequentially return elements from a Collection on which the iterator is defined.

In Fig. 4, iterators are used to sequentially access elements of extents which are Collections. In line 25, d_Iterator **`e`** is declared to access objects of the **`Employees`** extent. Member functions such as, **`next()`** and **`reset()`**, allow sequential navigation through these collections. Line 26 can be read "while there is a next element in the Collection associated with d_Iterator **`e`**, assign **`pe`** to reference this element and do lines 27 - 32." In the **`PrintDepts()`** function, a d_Iterator **`d`** is used to access objects from the **`Departments`** extent in order to print them.

6 Relationships

ODDL defines the relationships—or more precisely, associations—that can occur between the objects via ORN. Creation, change, and destruction of association links are performed in ODML via single-valued, object-valued attribute assignment operations and multi-valued, object-valued attribute insert, remove, and replace operations. Association semantics as defined by ORN are supported by ODML. That is, the ODBMS automatically maintains these semantics.

Variations on a *one-to-one* association may exist between the objects of two classes, whether the classes are the same or different. Specific objects may or may not be related via the association. For instance, Fig. 5 shows the potential for a one-to-one association between objects a and b but no association, or link, exists. When an association is defined between them, it is bi-directional. Fig. 6 below shows the association between a and b created by the operation a.B = &b (or b.A = &a).

Fig. 5

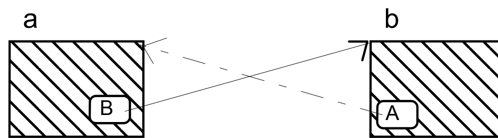

Fig. 6

The defined association is shown by the straight line, whereas the implied, or inverse, association is shown by the dashed line.

Variations on a *one-to-many* association may exist between the objects of two classes, whether the classes are the same or different. Again, specific objects may or may not be related via the association. Fig 7 shows the potential for a one-to-many association between objects a and b but no association, or link, exists. Fig. 8 shows an association between a and b which was created by the operation a.Bs.insert_element(&b) (or b.A = &a).

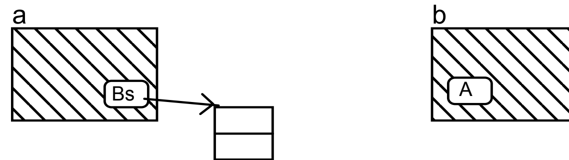

Fig. 7

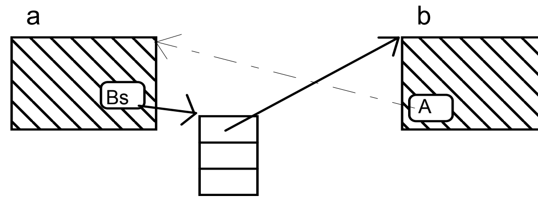

Fig. 8

A *many-to-many* association is illustrated in Figs. 9 and 10. Fig. 9 shows no association, or link, between objects a and b. Fig. 10 shows a link created by the operation a.Bs.insert_element(&b) (or b.As.insert_element(&a)) where the dashed line is the inverse association.

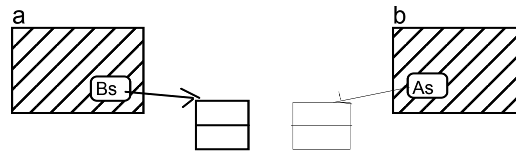

Fig. 9

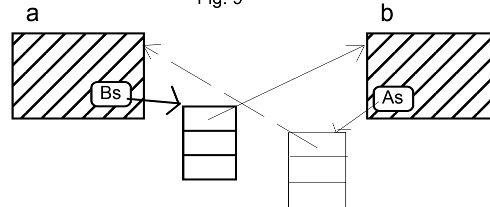

Fig. 10

In the above, association semantics as defined by ORN may apply. For instance, in the *one-to-one* association, if one element is deleted, the association link may be implicitly deleted. The same applies for the *one-to-many* and *many-to-many*.

Referring to Fig. 4 line 23, the assignment of **`Public Relations`** as the **`Dept`** for **`John Q. Doe`** creates an association link between the **`Public Relations`** and **`John Q. Doe`** objects. This link could have been set up another way. **`John Q. Doe`** (actually his address) could have been inserted into the attribute **`Employees`** for **`Public Relations`**. Regardless of the initial association set up, the OR+ makes changes to the inverse attribute to reflect the changes made to any object valued attribute. A graphical drawing of this association would be one to many. There must be one department for each employee, and there can be zero or many employees for each department.

The deletion of a temporary employee, i.e., **`pe->Delete();`** on line 28, causes more than just the deletion of the employee object. All of the relationship semantics associated with an employee, e.g., those listed below, are enforced. This means that many of the employee's associations are implicitly destroyed—perhaps, for example, the employee's involvement with a car pool—and related objects are implicitly deleted—most definitely a payroll record and perhaps assignments, children, a position, an address, and even a car pool. Association semantics related to an employee make an employee object a *complex* object.

- If an employee is deleted, all assignments for that employee and the employee's payroll record is also deleted.
- If an employee is deleted, the employee's address is deleted, unless it is also the address of another employee or customer. The employee's position is also deleted, unless this position is held by another employee, and all children of the employee are deleted unless the employee's spouse also works for the company. These semantics result from the **`'`** bindings.
- If an employee is deleted who is one of only two riders in a car pool, the car pool is deleted. This also occurs if this employee is not deleted but his relationship with the car pool is destroyed. That is, a car pool "is defined by" two or more riders.

The following are other association semantics related to an employee as defined by the ODDL in Fig. 2.

- A department can relate to zero, one, or more **(*)** employees and cannot be deleted if it has any employees (since the implicit destructibility binding for Department in the `association` is default and the **1** multiplicity would be violated on deletion of a department).
- An employee's relationship to a payroll record can never be destroyed.

The association semantics described above that result from the ORN given in an ODDL specification are explained in the ODDL User's Guide.

The assignment **`pe->Dept = pd`** on line 32 performs a association change from the perspective of an employee, i.e., it transfers an employee from customer support to public relations. Again, the inverse attribute **`Employees`** for both the customer support and public relation objects is automatically updated.

Suppose that instead of setting **`pe->Dept`** to **`pd`** here, the program had set **`pe->Dept`** to **`0`**, or null. This would cause a multiplicity exception since all employees must belong to **`1`** department.

7 Conclusion

ODML facilitates the retrieval and modification of objects and relationships in a database as defined by ODDL. Its two most important features are that it provides higher-level operations that treat objects as complex and composite objects based on ORN and it provides an ODMG-93 standard language for accessing and manipulating database objects.

References

1. R.G.G. Cattel, et al., *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, San Francisco, CA, 1997 (see Release 1.1 for description of future binding).
2. B.K. Ehlmann, G.A. Riccardi, and L.C. Dennis, "Representing Non-Inheritance Relationships in an Object-Oriented, Scientific Database," *Proceedings of the Sixth International Working Conference on Scientific and Statistical Database Management*, ETH Zurith, June 1992, pp. 99-109.
3. B.K. Ehlmann, L.C. Dennis, and G.A. Riccardi, "An Object-based Conceptual Model of a Nuclear Physics Experiments Database," *Nuclear Instruments & Methods in Physics Research*, Sect. A, Elsevier Science, North-Holland, **A325**, 1993, 294-308.
4. B.K. Ehlmann and G.A. Riccardi, "A Notation for Describing Aggregate Relationships in an Object-Oriented Data Model," *Applications of Databases—First International Conference Proceedings*, W. Litwin and T. Risch (Eds.), Lecture Notes in Computer Science 819, Springer-Verlag, 1994, pp. 62-77.
5. B.K. Ehlmann and G.A. Riccardi, "Object Relater Plus: A Practical Tool for Developing Enhanced Object Databases," *Proceedings 13th International Conference on Data Engineering*, Birmingham, UK, April 7-11, 1997, IEEE Computer Society Press, Los Alamitos, CA, pp. 412-421.
6. Progress Software Inc., ObjectStore interprise, 2009, www.objectstore.com/datasheet/index.ssp.

Appendix

```
#define RAISE(e, m)
//F: Raise exception e passing along message m.
//message given in RAISE of current exception

class Database {
public:
   enum access_status {not_open, read_write, read_only, exclusive}; //

   Database();
   //F: Initialize database as not opened and initialize database system if
   //   not previously initialized.

   access_status Status()
   //F: Return Status.

   void create(const char* database_name,  //external name for database
               unsigned int create_mode=0664, //Unix permissions for database
                                              //  (see chmod)
               int overwrite=0  // !0 => overwrite database if already exists
            );
   //F: Create and open database with read_write status.
   //   May raise ODML_DupOpen exception.
   //   Extension to ODMG-93 standard.

   void open(const char* database_name,    //external name for database
             access_status status = read_write,
             unsigned int create_mode = 0
                             // 0 => raise exception if database not present,
                             //!0 => create database if not present assigning
                             //      permissions for database (see chmod)
            );
   //F: Open database with read_write or read_only status.
   //   Creates database and opens it with given status, which must be
   //   read_write, if database not present and create_mode is not 0.
   //   May raise ODML_DupOpen exception.  Note that ODML can only enforce
   //   read_only on new operation since access to Database object not
   //   directly available.  Thus, left for Os to detect.

   void close()
   //F: Close database.

   void destroy();
   //F: Destroy database.
   //   May raise ODML_NotOpened and ODML_NotInTrans exceptions.
   //   Extension to ODMG-93 standard.

   void set_object_name(void* theObject, const char* theName);
   //F: Set theName as the name of theObject, thereby creating an entry point
   //   into the database via theName.
   //   Raises ODML_DupName exception if theName assigned to another object.
   //   May raise ODML_NotOpened, ODML_ReadOnly, and ODML_NotInTrans
   //   exceptions.
   //   Currently allows an object to be named twice and first parameter is
   //   not const due to OS limitations.  OS provision for type checking is
   //   not currently being used.

   void* lookup_object(const char* name);  //NULL -> object not found
   //F: Lookup, i.e., return pointer to, object with given name.
   //   May raise ODML_NotOpened and ODML_NotInTrans exceptions.
   //   Pointer to named object is not valid across transactions.
};  //Database

class Transaction {
public:
   Transaction()
   //F: Create a non-nested transaction.

   Transaction(Transaction* parent)
   //F: Create a nested transaction.

   ~Transaction();
   //F: Destroy transaction aborting it if still active.

   static Transaction* CurrTransPtr()
   //F: Return pointer to current transaction.

   void begin();
   //F: Begin a transaction.
   //   Raises ODML_DupBegin exception if called twice on transaction object
   //   without intervening abort/commit.  Assumes update.  Note that
```

```
   //   a transaction can span multiple databases.

   void commit();
   //F: Commit a tranaction.
   //   May raise ODML_NotInTrans or ODML_CardViolated exception.

   void abort();
   //F: Abort a transaction.
   //   May raise ODML_NotInTrans exception.

   void SetRXCmode()
   //F: Set RXC (Relationship eXChange) mode.
   //   In relationship exchange mode, violations of multiplicity lower
   //   bounds that occur when object-valued attributes are updated to
   //   make explicit relationship destructions are temporary ignored and no
   //   implicit deletes are initiated, i.e., a simple explicit destruction is
   //   allowed with no imposed ORN semantics.  The assumption is that these
   //   any violations are only temporary.  Appropriate multiplicity checks
   //   are deferred until transaction commit of parent to ensure multiplicities
   //   are not violated.  By default, a transaction is not in relationship
   //   exchange mode.
   //   The RXCmode of a parent transaction is inherited by a nested
   //   transaction.

   void ResetRXCmode()
   //F: Reset RXC (Relationship eXChange) mode.

   boolean RXCmode()
   //F: Return True iff in RXC (Relationship eXChange) mode.
};  //Transaction

#define TYPE_SPEC(type)
//F: Return type_spec of given type

void* operator new(size_t s,      //supplied by system
                   Database* db,  //-> Database object
                   type_spec t    //TYPE_SPEC(<class name>)
                  );
   //F: Create new object of type <class name> in Database db.
   //   May raise ODML_NotOpened, ODML_ReadOnly, and ODML_NotInTrans
   //   exceptions.

#define NEW(database, type)
//F: Create new persistent object in given database of given type with
//   subsequent constructor parameter list.
//   A convenient macro to specify new.  TYPE_SPEC does not have to be given
//   and the class name, i.e., type, does not have to be given twice.
//   An example of NEW to create a new employee whose constructor accepts
//      a string as the employee's name would be:
//   employee* p = NEW(myDB, employee)("Doe, John Q.");

class d_iterator {  //generic iterator--returns collection elements as void*
public:

   d_iterator(const d_collection& c);
   //F: Create iterator initialized to beginning of collection c.

   d_iterator(const d_rcollection& rc);
   //F: Create iterator initialized to beginning of d_rcollection rc.

   void reset()
   //F: Reset iterator to beginning of collection.

   int next(void* &p  //next element of collection or 0 if no more elements
           )
   //F: Get next element of collection as p returning 1 if successful, else 0.
   //   Assumes element of collection is a pointer.
};  //class d_iterator

class d_String {  //ODMG-93 standard d_String type
public:
   d_String();
   //F: Create empty d_String.

   d_String(const d_String& s);
   //F: Create d_String initialized to s.

   d_String(const char* p);
   //F: Create d_String initialized to character string p.

   ~d_String();
   //F: Destroy d_String.

   d_String& operator=(const d_String &s);
```

```
   //F: Assign s to d_String.

   d_String& operator=(const char* p);
   //F: Assign character string p to d_String.

   operator const char*() const;
   //F: Convert d_String to a char*.

   char& operator[](unsigned long i);
   //F: Reference i'th character of d_String where 0 <= i <= length of
   //   d_String-1.

   unsigned long length() const;
   //F: Return length of d_String.
};  //d_String

class d_ref { //reference (untyped), provides common base class for all
              // d_ref<TYPE>s, extension to standard
};  //class d_ref

class d_Ref_Any { //not needed for future binding implementation of ODMG,
   d_Ref_Any();
   //F: Create uninitialized d_Ref_Any.

   d_Ref_Any(void* p);
   //F: Create (convert p to) d_Ref_Any.

   d_Ref_Any(const d_Ref_Any& r);
   //F: Create (convert r to) d_Ref_Any.

   operator void*() const;
   //F: Convert d_Ref_Any to a void*.
};  //class d_Ref_Any

class d_collection {  //collection (untyped), allows access to
                    //   d_Collection<TYPE>s via generic d_iterator and
                    //   provides common base class for all
                    //   d_Collection<TYPE>s, extension to standard
public:
   int contains_address(void* ptr  //given address
                       );
   //F:  Return 1 if value of ptr is contained in collection, else return 0.
   //    A non-standard and somewhat dangerous function!
   //    Note that object can have different address depending on type cast.
   //    Thus, an object can be in a collection even though its address under
   //    a particular cast is not contained in a collection.
};  //class d_collection

inline int operator==(const d_String& sL, const d_String& sR)
//F: Return 1 if sL <op> sR, else 0, where <op> is relational operator.

inline int operator==(const d_String& sL, const char* pR)
//F: Return 1 if sL <op> pR, else 0, where <op> is relational operator.

inline int operator==(const char* pL, const d_String& sR)
//F: Return 1 if pL <op> sR, else 0, where <op> is relational operator.

inline int operator!=(const d_String& sL, const d_String& sR)

inline int operator!=(const d_String& sL, const char* pR)

inline int operator!=(const char* pL, const d_String& sR)

inline int operator< (const d_String& sL, const d_String& sR)

inline int operator< (const d_String& sL, const char* pR)

inline int operator< (const char* pL, const d_String& sR)

inline int operator<=(const d_String& sL, const d_String& sR)

inline int operator<=(const d_String& sL, const char* pR)

inline int operator<=(const char* pL, const d_String& sR)

inline int operator> (const d_String& sL, const d_String& sR)

inline int operator> (const d_String& sL, const char* pR)

inline int operator> (const char* pL, const d_String& sR)

inline int operator>=(const d_String& sL, const d_String& sR)

inline int operator>=(const d_String& sL, const char* pR)
```

```
inline int operator>=(const char* pL, const d_String& sR)

template <class TYPE>
class d_Iterator : public d_iterator {  //TYPE specific iterator
public:
   d_Iterator(const d_Collection<TYPE>& C);
   //F: Create d_Iterator initialized to beginning of typed collection.

   int next(TYPE &p  //next element of typed collection else 0 if no
                     //   more
                  );
   //F: Get next element of typed collection as p returning 1 if successful,
   //   else 0.
};  //class d_Iterator<TYPE>

template <class TYPE>
public:
   d_Ref();
   //F: Create uninitialized d_Ref<TYPE>.

   d_Ref(TYPE* p);
   //F: Create (convert p to) d_Ref<TYPE>.

   d_Ref(const d_Ref<TYPE>& r);
   //F: Create (convert r to) d_Ref<TYPE>.

   d_Ref(d_Ref_Any r);
   //F: Create (convert r to) d_Ref<TYPE>.

   operator TYPE*() const;
   //F: Convert d_Ref<TYPE> to a TYPE*.

   operator d_Ref_Any() const;
   //F: Convert d_Ref<TYPE> to a d_Ref_Any.

   TYPE* operator->() const;
   //F: Reference via d_Ref<TYPE>.

   TYPE& operator*() const;
   //F: Dereference d_Ref<TYPE>.
};  //class d_Ref<TYPE>

template <class TYPE>
class d_Collection : public d_collection {  //TYPE is assumed a pointer.
public:

   void insert_element(TYPE p);
   //F: Insert element p into collection.
   //   Raises err_coll_duplicate_insertion if duplicates disallowed and p
   //      is already in d_Collection.
   //   Raises err_coll_nulls if nulls disallowed and p is 0.

   void remove_element(TYPE p);
   //   May raise ODML_NotFound exception.

   TYPE remove_element();
   //F: Remove and return arbitrary element of collection, returning 0 if
   //   collection was empty.
   //   This is an extension to standard for implementation of ROVAs.
   //   Assumes TYPE is a pointer.

   int is_empty() const;
   //F: Return 1 if collection is empty, else 0.

   unsigned long cardinality() const;
   //F: Return number of elements in collection.

   int contains_element(TYPE p) const;
   //F: Return 1 if collection contains element p, else 0.
   //   NOTE:  See note for insert_element.

   void insert_element_first(TYPE p);
   //F: Insert element p into list as first element;
   //   Raises err_coll_not_supported if d_Collection not ordered.
   //   Raises err_coll_duplicate_insertion if duplicates disallowed and p
   //      is already in d_Collection.
   //   Raises err_coll_nulls if nulls disallowed and p is 0.

   void insert_element_last(TYPE p);
   //F: Insert element p into list as last element;
   //   Raises err_coll_not_supported if d_Collection not ordered.
   //   Raises err_coll_duplicate_insertion if duplicates disallowed and p
   //      is already in d_Collection.
```

```
   //   Raises err_coll_nulls if nulls disallowed and p is 0.

   void insert_element_before(TYPE p, unsigned int position);
   //F: Insert element p into list before element at given position.
   //   Raises err_coll_not_supported if d_Collection not ordered.
   //   Raises err_coll_duplicate_insertion if duplicates disallowed and p
   //      is already in d_Collection.
   //   Raises err_coll_nulls if nulls disallowed and p is 0.
   //   Raises err_coll_out_of_range if p not 0..cardinality().

   TYPE retrieve_element_at(unsigned int position) const;
   //F: Retrieve element at given position from collection.
   //   Raises err_coll_not_supported if d_Collection not ordered.
   //   Raises err_coll_out_of_range if p not 0..cardinality()-1.

   int find_element(TYPE p, unsigned int &position) const;
   //F: Find position of given element p in collection, returning 1 if
   //   found, else 0.
   //   position is 0..cardinality()-1 if found.  position for an
   //      unordered collection indicates only number of elements that
   //      preceded p in search of collection.
   //   NOTE:  See note for insert_element
};  //class d_Collection<TYPE>

template <class TYPE>
class d_Set :public d_Collection<TYPE> {  //unordered, unique collection
public:
   d_Set();
   //F:  Create d_Set<TYPE> as a d_Collection<TYPE> with set behavior.
};  //class d_Set<TYPE>
template <class TYPE>
class d_List :public d_Collection<TYPE> {  //ordered collection
public:
   d_List();
   //F:  Create d_List<TYPE> as a d_Collection<TYPE> with list behavior.
};  //class d_List<TYPE>
```